PyTorch

Students: Ambareesh Shyam Sundar, Ignacio Bartol, Matias Vizcaino
Mentors: Sahdev Zala (IBM), Thara Palanivel (IBM)

Georgia Tech Open Source Program Office

Project Overview

An **open-source** machine learning framework that accelerates the path from research prototyping to production deployment.

End User

- Deep learning primitives, NN Layers, activation & loss functions, optimizers.
- Models are Python code, Autograd and "eager mode" for dynamic model architectures.
- Production deployment: torch.compile, TorchServe, quantization.

Developer

- 70+ repositories. pytorch is the core one.

Codebase structure:

- ❑ c10: Core library files that work everywhere.
- ❑ aten: C++ tensor library for Pytorch.
- ❑ torch: The actual PyTorch library (jit, autograd, dynamo, etc.)
- ❑ Tools, tests, caffe2, etc.

PyTorch has a foundation governance, with a small set of module maintainers, core maintainers, and 3390 contributors (as of July 19th 2024). Each PR needs to pass 167 checks with many tests for each.

Goals and Milestones

1st milestone:

1. Contributor License Agreements signed (pytorch and tutorials repos)
2. Familiarity with few different repositories on github.com/pytorch
3. Setup and verify PyTorch dev environment and installation.
4. Develop understanding of open source and how it works.
5. Walk through the PyTorch dev and contribution docs.
6. Have at least one PR created.

2nd milestone:

1. Participation in the PyTorch Docathon - (optional but recommended)
2. More than one PRs created.
3. (As needed) Participated in the PyTorch project Office Hours. Developed a good understanding of the PyTorch core repo and other repos in the ecosystem. Demonstrate it via a presentation to mentors during the 2nd milestone.

Open-source Outcomes

Linux Foundation EasyCLA (pytorch repo) – Facebook/Meta CLA (tutorials repo)

Lessons learned:
PyTorch development environment setup -- how to setup the C++ toolchain, Conda environment, running tests and linting locally, etc.

Focus on one specific PyTorch module, learn about how it works, the maintainers and possible contributions by filtering open issues with specific labels on that module.

If your PR gets rebased make sure to pull the changes and before pushing make sure (HEAD -> main, origin/main, origin/HEAD) are in the right commit. If not the next commit can be super messy and mess up your PR.

Highlights and Accomplishments – PyTorch Docathon

Timeline:
June 4: Kick-off; **June 4 - 16**: Submissions and Feedback; **June 17 - 18**: Final Reviews.

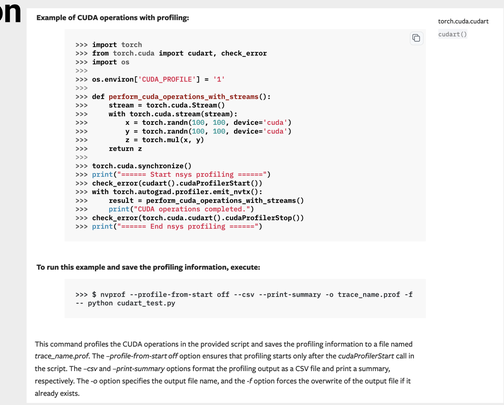
Goal of the Docathon:
To quickly start to collaborate on the PyTorch project by working on improving documents or documenting functions from pytorch or improving the tutorials.

Lessons learned:
Very active and responsive maintainers during this period, so you can clear your questions and get your PRs reviewed quite fast (1-3 days).
Working on documentation can be fun as it require you to understand the specific module, explain it in a nice way and play by creating some (working and useful) examples.

Highlights of the Docathon H1 2024:
Submitted 7 PRs (6 ''medium' and 1 "easy" fixes).
Got 2nd place for the 4 PRs that got approved before Jun. 20th
Learnt how to build docs using Sphinx and how automodule and autosummary works.
PyTorch has strict guidelines for writing docs that can be found here.

Some highlights of the PRs:
Pytorch/pytorch repo:
"Created docs for cudart function in torch.cuda #128741"
The function retrieves the CUDA runtime API module by calling to _lazy_init(), therefore all the Raises exceptions are from the function is being called.

Creating a useful example was particularly challenging and something I got comments on the review as well. (see image)



Example of CUDA operations with profiling:

torch.cuda.cudart

cudart()

```
>>> import torch
>>> from torch.cuda import cudart, check_error
>>> import os
>>>
>>> os.environ['CUDA_PROFILE'] = '1'
>>>
>>> def perform_cuda_operations_with_streams():
>>>     stream = torch.cuda.Stream()
>>>     with torch.cuda.stream(stream):
>>>         x = torch.randn(100, 100, device='cuda')
>>>         y = torch.randn(100, 100, device='cuda')
>>>         z = torch.mul(x, y)
>>>     return z
>>>
>>> torch.cuda.synchronize()
>>> print("====== Start nsys profiling ======")
>>> check_error(cudart().cudaProfilerStart())
>>> with torch.autograd.profiler.emit_nvtx():
>>>     result = perform_cuda_operations_with_streams()
>>>     print("CUDA operations completed.")
>>> check_error(torch.cuda.cudart().cudaProfilerStop())
>>> print("====== End nsys profiling ======")
```

To run this example and save the profiling information, execute:

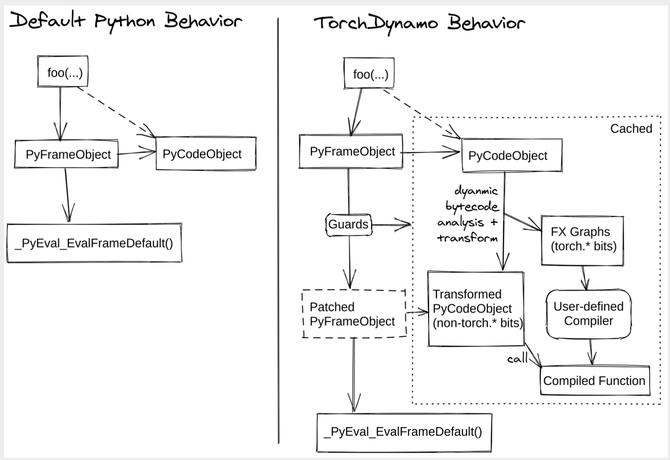
```
>>> $ nvprof --profile-from-start off --csv --print-summary -o trace_name.prof -f
-- python cudart_test.py
```

This command profiles the CUDA operations in the provided script and saves the profiling information to a file named *trace_name.prof*. The *–profile-from-start off* option ensures that profiling starts only after the *cudaProfilerStart* call in the script. The *–csv* and *–print-summary* options format the profiling output as a CSV file and print a summary, respectively. The -o option specifies the output file name, and the *-f* option forces the overwrite of the output file if it already exists.

Pytorch/tutorials repo:
Improved the torch.compile tutorial on how to use torch.compile in nested modules and functions. Also, I added a section for "Best practices" to debug if there is a compilation problem and how to disable a function from being compiled (along with its children's functions) using torch.compiler.disable.

Highlights and Accomplishments – torch.compile and Dynamo

Torch Dynamo: JIT compiler for PyTorch code



Intercept Python bytecode before it is passed to the virtual machine

Represents each compiled function with a computation graph which it tries to optimize just-in-time

Creates graph breaks when the compiler encounters something it cannot handle

Fake Tensors: used to represent the shape and type of tensors present in the compilation graph to allow for further optimizations

Issues: errors with certain FakeTensor operations not being type-checked properly

My learning process:

- Learn about Torch Dynamo internals (e.g. how the compilation graph is constructed and optimized)
- Examine the Python implementation of the FakeTensor subclass and the C++ implementation of Tensor operations
- Modify the relevant overloaded implementations to include type-checking and raise an exception if the types being passed in are invalid

PR Raised: Fixed the issue of converting user-defined objects to strings in TorchDynamo by adjusting the call_str method.

Explored PyTorch XLA for TPU Training: Experimented with setting up a TPU environment for more efficient training and inference of models through Google Colab but encountered limitations.

Future Work

Continue contributing to PyTorch and related libraries, especially in the following areas:

- Optimizing Torch Dynamo and allowing it to be compatible with more of the PyTorch library and ecosystem (e.g. TensorDict, Python generators)
- Contributing to the core c10/ATen C++ libraries to rewrite and refactor legacy code and improve individual kernel implementations
- Contribute to update the tutorials as many of them are outdated given the multiple releases of PyTorch over the years.

Georgia Tech